Hybrid Programming with OpenMP and MPI

Kushal Kedia (kushal@mit.edu)

Project Report: 18.337 Applied Parallel Programming

May 13, 2009

1 Introduction

The basic aims of parallel programming are to decrease the runtime for the solution to a problem and increase the size of the problem that can be solved. The conventional parallel programming practices involve a a pure OpenMP implementation on a shared memory architecture (Fig. 1) or a pure MPI implementation on distributed memory computer architectures (Fig. 2). The largest and fastest computers today employ both shared and distributed memory architecture (Fig. 3). This gives a flexibility in tuning the parallelism in the programs to generate maximum efficiency and balance the computational and communication loads in the program. A wise implementation of hybrid parallel programs utilizing the modern hybrid computer hardware can generate massive speedups in the otherwise pure MPI and pure OpenMP implementations. A brief introduction to OpenMP, MPI and Hybrid programs is given in the following subsections.

OpenMP

OpenMP is an API (Application Programming Interface) for writing multithreaded applications. It takes the advantage of multicore (multiple processors) on a single memory. They are a part of standard libraries for C, C++ and

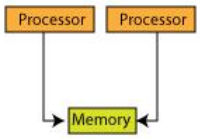

Figure 1: Shared memory, ideal for OpenMP

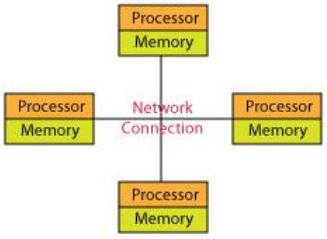

Figure 2: Distributed memory, ideal for MPI

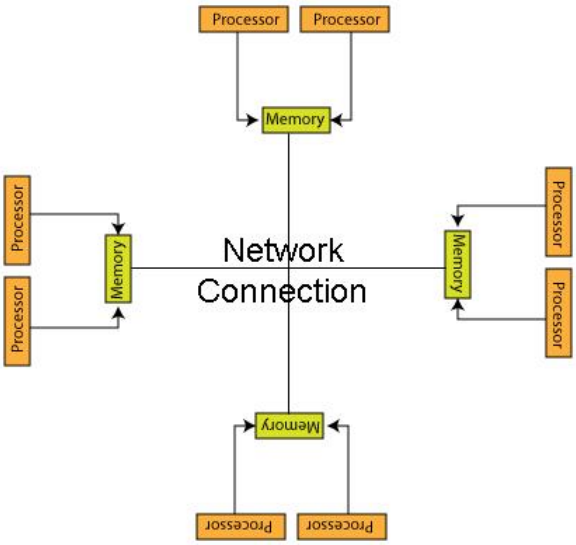

Figure 3: Shared and distributed memory, modern computer architecture. Suitable for MPI and hybrid implementations

FORTRAN. The pros of pure OpenMP implementations include easiness to implement, low latency and high bandwidth, implicit communication and dynamic load balancing. However, it can be used on shared memory machines only, can be scaled only withing one node and has a random threading order.

MPI (Message Passing Interface)

The fundamental limitation of OpenMP, that it can be used on shared memory machine only, is solved with MPI. MPI is portable to distributed and shared memory machines. It scales to multiple shared memory machines with no data placement problems. However, it is difficult to develop and debug, it has high latency and low bandwidth and explicit communication is required. Even if there are multiple processors sharing a memory, MPI implementations will treat each processor as separate and explicit communications between them is needed, despite the fact that memory is being shared. Like OpenMP, C, C++ and FORTRAN bindings of MPI are easily available.

Hybrid concept

Hybrid programming is an attempt to utilize the best from both the above scenarios. Hybrid programming may not always be beneficial, since it depends heavily on the problem statement and computational and communication loads. However, if it works on a problem, it can give considerable speedups and better scaling. It is a modern software trend for the current hybrid hardware architectures. The basic concept is to use MPI across the nodes and OpenMP within the node. This avoids the extra communication overhead with MPI within a single node.

In this project, a hybrid implementation of a Laplace solver is compared with pure MPI implementation. The programming language C is used for this work. Pure MPI program and Hybrid programs were developed for this work.

2 Problem Statement

A simple elliptic Laplace equation is solved. The equation is mathematically expressed as

$$\frac{\partial u}{\partial x^2} + \frac{\partial u}{\partial y^2} = 0 \tag{1}$$

where $u = u(x, y)$is an unknown scalar potential subjected to the following boundary conditions:

$$u(x, 0) = \sin(\pi x) \qquad 0 \leq x \leq 1$$

$$u(x, 1) = \sin(\pi x)e^{-x} \qquad 0 \leq x \leq 1$$

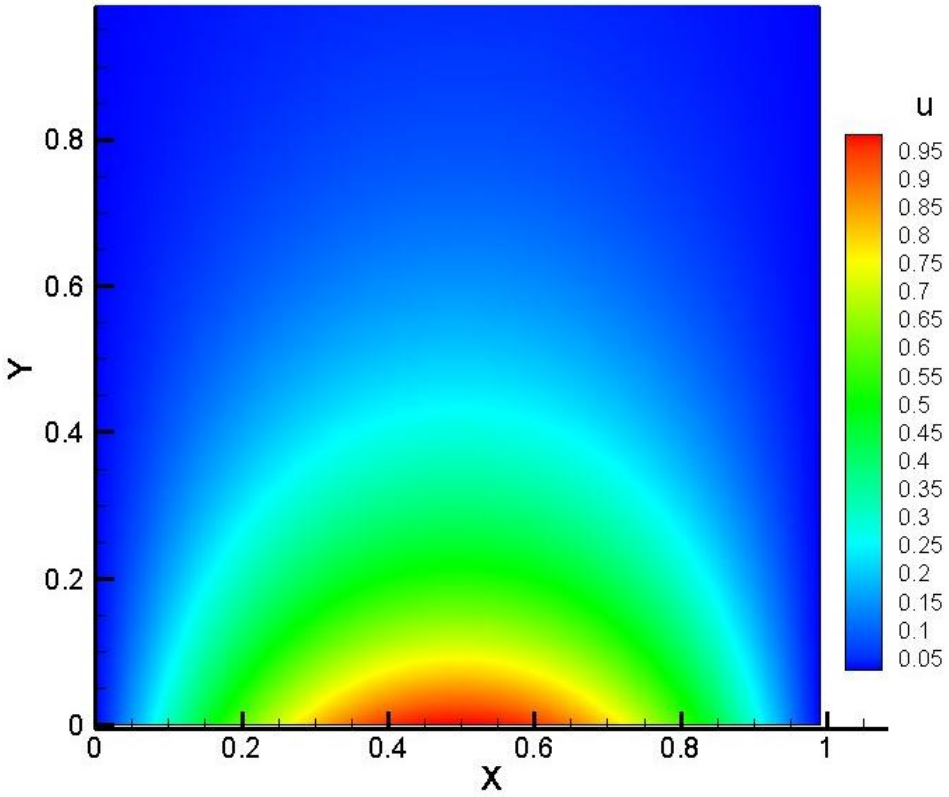

Figure 4: Solution to the defined problem

$$u(0,y) = u(1,y) = 0 = 0 \qquad 0 \leq y \leq 1$$

Such boundary value problems are very common in physical problems. Example of a physical problem having similar mathematical description is a two-dimensional steady-state heat equation. This particular problem statement is chosen here since its analytical solution is easy to obtain. It can be expressed as

$$u(x,y) = \sin(\pi x)e^{-\pi y}; \qquad 0 \leq x \leq 1; \quad 0 \leq y \leq 1 \tag{2}$$

Equation 1 is solved numerically using finite differences and the convergence test will be based on this analytical solution. The solution to this problem is shown in Fig. 4. The codes are validated by comparing it to this solution. This problem can be physically interpreted as a steady state heat transfer problem in a room with a heater at one of the walls. Such problems have similar mathematical nature.

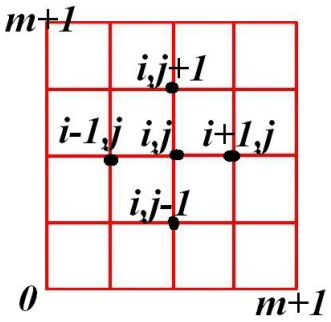

Figure 5: Schematic of the two-dimensional grid.

3 Numerical Scheme

A uniform two-dimensional grid is used. Equation 1 can be discretized, using central differences, in an algebraic equation

$$u_{i,j}^{n+1} = \frac{u_{i-1,j}^{n} + u_{i+1,j}^{n} + u_{i,j-1}^{n} + u_{i,j+1}^{n}}{4} \qquad i = 1, ..., m; \quad j = 1, ..., m \tag{3}$$

where $n+1$denotes the next iteration and $n$denotes the current iteration, while

$$u_{i-1,j}^{n} = u^{n}((i-1)h, jh)$$

where $h = \frac{1}{m+1}$is the grid resolution (Note: A uniform grid resolution in both the directions is assumed). This scheme is well known as the Jacobi iterative scheme for Laplace equation. A all zero-value initial guess of the solution is taken and Jacobi iterations are performed until convergence is obtained. A better initial solution could be chosen to improve the convergence rate, but the purpose of this work is to have longer running codes, to measure wall clock time of the code for performance analysis. The grid schematic is shown in Fig. **??**.

4 Grid Decomposition

The parallel distributed uniform grid is shown in Fig. 6. The thick red lines show the decomposition of the grid among 4 processors. Since the grid is uniform, and two-dimensional, there are typically two choices of domain decomposition , one-dimensional or two-dimensional. Assuming p processors are used, the computational domain is split in $p$horizontal strips as shown in Fig. 6. To have better load balancing, each processor is given an equal chunk of the grid, approximately $m \times m/p$ size each. Each processor communicates with the processor above (north) and below (south) (see Fig. 6) after every iteration, to compute the solution using the discretization given by Eq. 3.

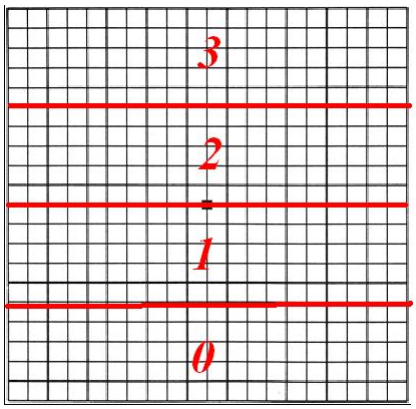

Figure 6: The uniform two-dimensional grid distributed on 4 processors

5 Computational Resource

Pharos cluster (a shared and distributed memory architecture), used by the Reacting Gas Dynamics Laboratory (Dept. of Mechanical Engineering, MIT), is used for the parallel simulations. Pharos consists of about 60 Intel Xeon - Harpertown nodes, each consisting of dual quad-core CPUs (8 processors) of 2.66 GHz speed.

The pure MPI implementation takes $p$processors such that few or none of them may belong to the same node. This implies that there are MPI calls between all the $p$processors. The hybrid implementation takes $p$processors in such a way that there are $x$nodes and $4x = p$. This implies that there are MPI calls between $x$nodes and OpenMP calls between 4 processors on each node which share the memory. These two configurations are reported as "MPI" and "Hybrid" in the following section.

Hybrid Execution Scenario

A single MPI process is launched on each SMP node (Symmetric Multi-Processor - with shared memory). Each process spawns 4 threads on each of these SMP nodes. After every parallel OMP iteration within each SMP node, the master thread of each SMP nodes communicates with other master threads of other MPI nodes using MPI calls. Again the iteration in OpenMP within each node is carried out with threads until it is complete. This scenario is schematically shown in Fig. 7.

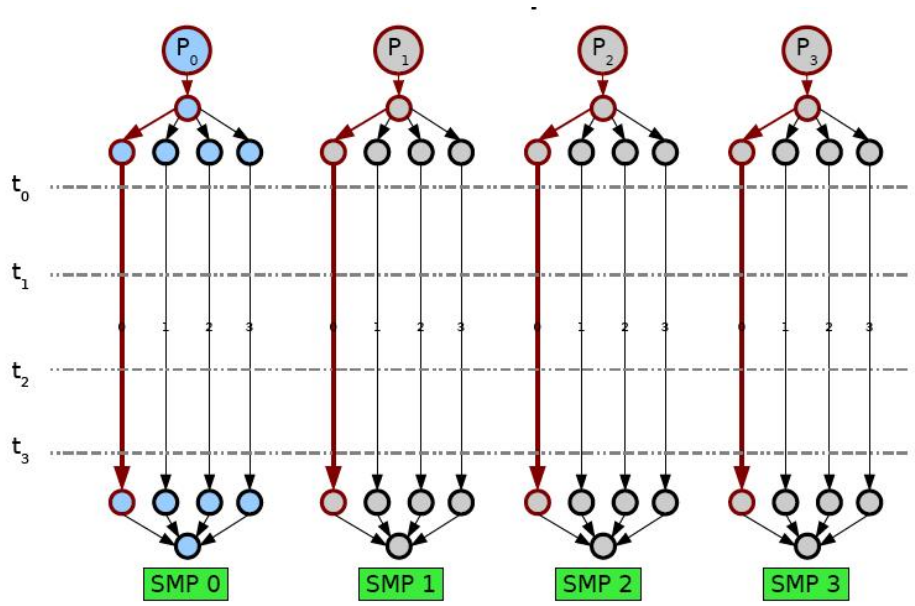

Figure 7: Hybrid Execution Scenario, figure adopted from Reference 5

6 Results

The time per iteration of the Jacobi schemes are reported in the results. The time per iteration reported are averaged over the number of processors and averaged over few runs of the same configuration.

Figure 8shows the performance of pure MPI and hybrid schemes for a 2500×2500 grid. The pure MPI scheme is faster than the hybrid scheme. The difference is high for lesser number of processors and it reduces as the number of processors increase. This may be due to the large grid size. There may be cache misses when the grid size per processor is larger. In the hybrid scheme, each node gets a bigger chunk of grid, which is solved with OpenMP threads, unlike pure MPI where all the processors get a smaller chunk of grid. As the number of processors increases, the grid size per node for hybrid scheme also decreases, thereby making the code run faster and scale better. However, the speed never matches that of the pure MPI schemes.

Figure 9shows the performance of pure MPI and hybrid schemes with increasing grid size. The number of processors are kept constant at 20. As expected, each of the schemes take large time as the grid size increases due to more computations. Even in this case, the pure MPI scheme work better than hybrid scheme. As the size of the grid increases, the hybrid schemes perform slower than pure MPI schemes, again probably due to cache misses, since each node gets a much bigger chunk of the grid for computation.

Hybrid schemes dont always work better than pure MPI schemes, as we

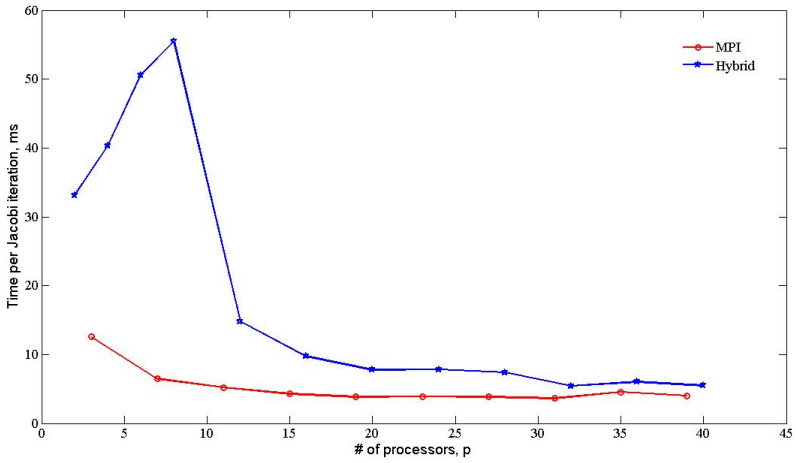

Figure 8: Performance comparison of pure MPI and hybrid implementations for a fixed grid size of 2500×2500

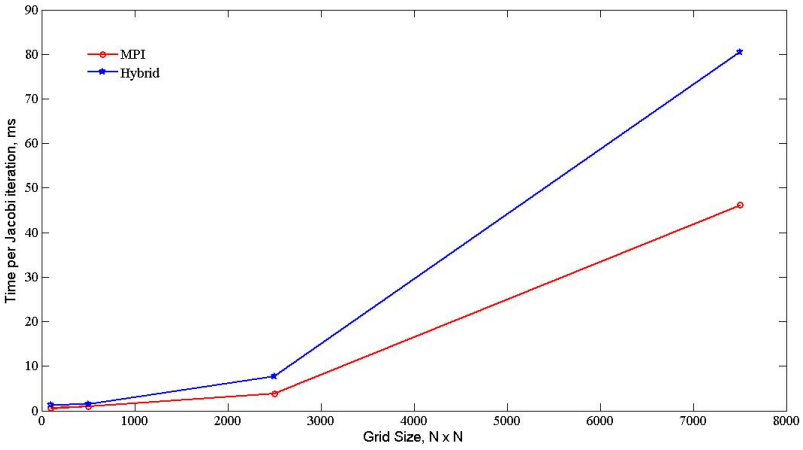

Figure 9: Performance comparison of pure MPI and hybrid implementations for a fixed number of processors, $p = 20$

observe. They depend on the nature of the computation. This fact is observed by many hybrid programmers, and is not a surprising result. The possible reasons for this are discussed in the conclusions.

7 Conclusions

Hybrid programming is compared to pure MPI program of a standard Laplace solver. It is seen that hybrid program is slower compared to the pure MPI program for different grid sizes and different number of processors.

Hybrid programming is an attempt to maximize the best from both OpenMP and MPI paradigms. However, it does not always imply better performance. This may be due to a number of reasons. OpenMP has less scalability due to implicit parallelism while MPI allows multi-dimensional blocking. All threads are idle except one while MPI communication. There is a thread creation overhead. The chances of cache miss problems increase due to data placement problem and larger dataset. Pure OpenMP performs slower than pure MPI within a node due to lack of optimized OpenMP libraries. Hybrid programs tend to work better when the communication to computation ratio is high. In the Laplace problem considered in this project, there is communication only at the boundary grid points of each processor and the cache miss during the computation may be a overhead over this communication.

Converting pure MPI codes into Hybrid codes is not a particularly difficult task. It is worth a try because if the nature of the problem (computation and communication) is such that it can allow for faster hybrid codes, the speedups can be huge. There has been both positive and negative experiences with hybrid codes and the programmer has to program and verify to decide to choose hybrid programming over pure MPI. Currently, there are very few benchmarked results for hybrid programs, and all of them are very problem specific.

8 Bibliography

1. William Gropp, Ewing Lusk, and Anthony Skjellum , **Using MPI: Portable Parallel Programming with the Message-Passing Interface**, MIT press, 1999.

2. William Gropp, Ewing Lusk, and Rajeev Thakur, **Using MPI-2: Advanced Features of the Message-Passing Interface**, MIT press, 1999.

3. Barbara Chapman, Gabriele Jost and Ruud van der Pas, **Using OpenMP Portable Shared Memory Parallel Programming**, MIT press, 2007.

4. Yun (Helen) He and Chris Ding , **Hybrid OpenMP and MPI Programming and Tuning**, Lawrence Berkeley National Laboratory , www.nersc.gov /nusers/services/training/classes/NUG/Jun04/NUG2004_yhe_hybrid.ppt

5. B. Estrade, High Performace Computing Workshop, **Hybrid Programming with MPI and OpenMP**, www.hpc.lsu.edu/training/tutorials /presentations/intro-hybrid-mpi-openmp.pdf

6. Sebastian von Alfthan, CSC – the Finnish IT Center for Science, PRACE summer school 2008, **Introduction to Hybrid Programming**, http://www.prace-project.eu/hpc-training /prace-summer-school/hybridprogramming.pdf

7. Blaise Barney, Lawrence Livermore National Laboratory, **Introduction to Parallel Computing**, https://computing.llnl.gov/tutorials/parallel_comp/

Appendix A - Pure MPI code

```
/* PURE MPI LAPLACE SOLVER - SEE THE REPORT FOR EXACT PROBLEM
DEFINITION*/

#include "mpi.h"
#include <stdio.h>
#include <math.h>
#include <time.h>

/* problem parameters*/
#define N 2500 // no of gridpoints .. Current code validated for square
grid only.. Nx = Ny = N
#define ITER 200 // total no of iterations to be performed


#define MASTER 0 // id of the first processor

/*define message tags for sending and receiving MPI calls*/
#define BEGIN 1 // message type
#define DONE 2 // message type
#define STH 3 // message type
#define NTH 4 // message type

/* define the number of threads to spawn */

int main(int argc,char *argv[])
{
  int myid, nprocs;
  MPI_Status status;

  int Nx = N;
  int Ny = N;

  /*functions*/
  void init();  //initializing the solution
  void save();  //write the solution in a file
  float errorcheck(); // check the error compared to analytical
solution

  float u[2][Ny][Nx]; //variable to solve

  int min_rows, overflow;

  int slave, south, north; //processor identity and its neighbours'
identity
  int num_rows; // num_rows for each processor
  int destination, source; // for convenient msg passing
  int msg;
  int base;
  int i,j,k;

  int start, end;  //starting and ending j indices of each chunk of row
for each processor - row wise domain distribution
  float eps=0.1;
```

```
  int count;

  double start_time,end_time;


  /*Initialize the MPI environment*/
  MPI_Init(&argc,&argv);

  /*current id and total no of processes*/
  MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  start_time = MPI_Wtime();

  /*Master Task. Divides the data among processors and collects and
collocates data back. No computation performed by master*/
  if(myid==MASTER)
    {
      printf("Gridsize Nx x Ny= %d x %d; \t ; \t Max Iterations= %d;
\n",Nx, Ny, ITER);
      printf("INitializing the solution \n");
      printf("\n");
      init(Nx, Ny, u);

      /*only nprocs-1 processors are performing actual computation.
Master is just co-ordinating*/
      min_rows = Ny/(nprocs-1);
      overflow = Ny%(nprocs-1);
      base=0;

      for(i=1;i<=nprocs-1;i++)
      {
        if(i<=overflow)
            num_rows=min_rows+1;
        else
            num_rows=min_rows;

        /*processor 0 is our Master. Processors 1, 2, 3..... till
nprocs-1 are the actual working processors*/
        if(i==1)
          south=0; //no south neighbour for the first processor
        else
          south=i-1;
        if(i==nprocs-1)
          north=0; //no north neighbour for the last processor
        else
          north=i+1;

        destination = i;
        slave = i;
        msg = BEGIN;

        /* Send the required information to each node */
        MPI_Send(&slave, 1, MPI_INT, destination, msg, MPI_COMM_WORLD);
        MPI_Send(&base, 1, MPI_INT, destination, msg, MPI_COMM_WORLD);
        MPI_Send(&num_rows, 1, MPI_INT, destination, msg,
MPI_COMM_WORLD);
        MPI_Send(&south, 1, MPI_INT, destination, msg, MPI_COMM_WORLD);
```

```
        MPI_Send(&north, 1, MPI_INT, destination, msg, MPI_COMM_WORLD);
        MPI_Send(&u[0][base][0], num_rows*Nx, MPI_FLOAT, destination,
msg, MPI_COMM_WORLD);

        printf("Sent to= %d; \t j_index= %d; \t num_rows= %d; \t
south_neighbour= %d; \t north_neighbour=%d\n",
             destination,base,num_rows,south,north);

        base += num_rows;

      }

      /* Collecting and collocating the results */
      for(i=1;i<=nprocs-1;i++)
      {
        source = i;
        msg = DONE;
        MPI_Recv(&base, 1, MPI_INT, source, msg, MPI_COMM_WORLD,
&status);
        MPI_Recv(&num_rows, 1, MPI_INT, source, msg, MPI_COMM_WORLD,
&status);
        MPI_Recv(&u[0][base][0], num_rows*Nx, MPI_FLOAT, source, msg,
MPI_COMM_WORLD, &status);
      }

      /* WRITE FINAL SOULTION*/
      // save(Nx, Ny, &u[0][0][0], "output.dat");


    }
/**************** End of master code *********************************/



/* Slaves code */
  if (myid != MASTER)
    {

      for(k=0;k<2;k++)
      for (i=0; i<Nx; i++)
        for (j=0; j<Ny; j++)
          u[k][j][i] = 0.0;


      /* Receive data from MASTER*/

      source = MASTER;
      msg = BEGIN;
      MPI_Recv(&slave, 1, MPI_INT, source, msg, MPI_COMM_WORLD,
&status);
      MPI_Recv(&base, 1, MPI_INT, source, msg, MPI_COMM_WORLD,
&status);
      MPI_Recv(&num_rows, 1, MPI_INT, source, msg, MPI_COMM_WORLD,
&status);
      MPI_Recv(&south, 1, MPI_INT, source, msg, MPI_COMM_WORLD,
&status);
```

```
      MPI_Recv(&north, 1, MPI_INT, source, msg, MPI_COMM_WORLD,
&status);
      MPI_Recv(&u[0][base][0], num_rows*Nx, MPI_FLOAT, source, msg,
MPI_COMM_WORLD, &status);


      for (i=0; i<Nx; i++)
      u[1][0][i]=u[0][0][i];

      for (i=0; i<Nx; i++)
      u[1][Ny-1][i]=u[0][Ny-1][i];

      for (j=0; j<Ny; j++)
      u[1][j][0]=u[0][j][0];

      for (j=0; j<Ny; j++)
      u[1][j][Nx-1]=u[0][j][Nx-1];

      if (base==0)
      start=1; // do not include bottom row or row 0 which is the
boundary
      else
      start=base;

      if (base+num_rows==Ny)
      end= base + num_rows-2;  //do not include top row which is also
the boundary
      else
      end = base + num_rows-1;

      k=0;

      for(count=0; count<=ITER; count++)
      {


        if (south != 0)
          {
            MPI_Send(&u[k][base][0], Nx, MPI_FLOAT, south, NTH,
MPI_COMM_WORLD);
            MPI_Recv(&u[k][base-1][0], Nx, MPI_FLOAT, south, STH,
MPI_COMM_WORLD, &status);
          }
        if (north != 0)
          {
            MPI_Send(&u[k][base+num_rows-1][0], Nx, MPI_FLOAT, north,
STH, MPI_COMM_WORLD);
            MPI_Recv(&u[k][base+num_rows][0], Nx, MPI_FLOAT, north,
NTH, MPI_COMM_WORLD, &status);
          }


        for (j = start; j <= end; j++)
          for (i = 1; i <= Nx-2; i++)
            u[1-k][j][i] = (u[k][j][i+1]+u[k][j][i-
1]+u[k][j+1][i]+u[k][j-1][i])*0.25;
```

```
        k = 1 - k;
      }


      eps=errorcheck(start, end, Nx, Ny, &u[k][0][0]);


      /*Send data back to master*/
      destination=MASTER;
      msg=DONE;
      MPI_Send(&base, 1, MPI_INT, destination, msg, MPI_COMM_WORLD);
      MPI_Send(&num_rows, 1, MPI_INT, destination, msg,
MPI_COMM_WORLD);
      MPI_Send(&u[k][base][0], num_rows*Nx, MPI_FLOAT, destination,
msg, MPI_COMM_WORLD);



      printf("Processor number: %d; eps = %6.12f \n", myid, eps);

    }

  end_time=MPI_Wtime();

  printf ( "Processor number: %d; Total Elapsed time for pure MPI
implementation is  %f  seconds\n", myid, end_time-start_time);


  MPI_Finalize();

}


/*Subroutines*/

void init(int nx, int ny, float *u)
{
  int i,j;
  double hx, hy;

  hx=1.0/(nx-1);
  hy=1.0/(ny-1);


  for (j = 1; j <= ny-2; j++)
    for (i = 1; i <= nx-2; i++)
      *(u+j*nx+i) = (float)(0);  //all the interior points are set to 0

  j=0;
  for (i = 0; i <= nx-1; i++)
    *(u+j*nx+i) = sin(3.14*i*hx);  //bottom boundary

  j=ny-1;
  for (i = 0; i <= nx-1; i++)
    *(u+j*nx+i) = exp(-3.14)*sin(3.14*i*hx);  //top  boundary

  i=0;
```

```
  for (j = 0; j <= ny-1; j++)
    *(u+j*nx+i) = (float)(0);  //left boundary

  i=nx-1;
  for (j = 0; j <= ny-1; j++)
    *(u+j*nx+i) = (float)(0);  //right boundary

}

void save(int nx, int ny, float *u1, char *output)
{
  int i, j;
  FILE *f_out;


  f_out = fopen(output, "w");

  for (i = nx-1; i >=0; i--)
    for (j = 0; j <= ny-1; j++)
      fprintf(f_out, "%d %d %6.12f\n", i, j, *(u1+j*nx+i));
  fclose(f_out);
}

float errorcheck(int start, int end, int nx, int ny, float *u)
{
  int i,j;
  float sum = 0.0;
  float exact;

  double hx, hy;

  hx=1.0/(nx-1);
  hy=1.0/(ny-1);

  for (j = start; j <= end; j++)
    {
      for (i = 1; i <= nx-2; i++)
      {
        exact = (sin(3.14*i*hx))*exp(-3.14*j*hy); //exact solution at
the point
        sum = sum + (exact - *(u+j*nx+i))*(exact - *(u+j*nx+i));
//relative error
      }
    }

  return sqrt(sum);
}
/*END OF PROGRAM*/
/************************************************************************
***************************************/
```

Appendix B - Hybrid code

```
/* HYBRID LAPLACE SOLVER - SEE THE REPORT FOR EXACT PROBLEM
DEFINITION*/

#include "mpi.h"
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <omp.h>

/* problem parameters*/
#define N 2500 // no of gridpoints .. Current code validated for square
grid only.. Nx = Ny = N
#define ITER 200 // total no of iterations to be performed


#define MASTER 0 // id of the first processor

/*define message tags for sending and receiving MPI calls*/
#define BEGIN 1 // message type
#define DONE 2 // message type
#define STH 3 // message type
#define NTH 4 // message type

/* define the number of threads to spawn */
#define NUM_THREADS 4


int main(int argc,char *argv[])
{
  int myid, nprocs;
  MPI_Status status;

  int Nx = N;
  int Ny = N;

  /*functions*/
  void init();  //initializing the solution
  void save();  //write the solution in a file
  float errorcheck(); // check the error compared to analytical
solution

  float u[2][Ny][Nx]; //variable to solve

  int min_rows, overflow;

  int slave, south, north; //processor identity and its neighbours'
identity
  int num_rows; // num_rows for each processor
  int destination, source; // for convenient msg passing
  int msg;
  int base;
  int i,j,k;

  int start, end;  //starting and ending j indices of each chunk of row
for each processor - row wise domain distribution
```

```
  float eps=0.1;

  int count;

  double start_time,end_time;


  /*Initialize the MPI environment*/
  MPI_Init(&argc,&argv);

  /*current id and total no of processes*/
  MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  start_time = MPI_Wtime();
  omp_set_num_threads(NUM_THREADS);


  /*Master Task. Divides the data among processors and collects and
collocates data back. No computation performed by master*/
  if(myid==MASTER)
    {
      printf("Gridsize Nx x Ny= %d x %d; \t ; \t Max Iterations= %d;
\n",Nx, Ny, ITER);
      printf("INitializing the solution \n");
      printf("\n");
      init(Nx, Ny, u);

      /*only nprocs-1 processors are performing actual computation.
Master is just co-ordinating*/
      min_rows = Ny/(nprocs-1);
      overflow = Ny%(nprocs-1);
      base=0;

      for(i=1;i<=nprocs-1;i++)
      {
        if(i<=overflow)
            num_rows=min_rows+1;
        else
            num_rows=min_rows;

        /*processor 0 is our Master. Processors 1, 2, 3..... till
nprocs-1 are the actual working processors*/
        if(i==1)
          south=0; //no south neighbour for the first processor
        else
          south=i-1;
        if(i==nprocs-1)
          north=0; //no north neighbour for the last processor
        else
          north=i+1;

        destination = i;
        slave = i;
        msg = BEGIN;

        /* Send the required information to each node */
        MPI_Send(&slave, 1, MPI_INT, destination, msg, MPI_COMM_WORLD);
```

```
        MPI_Send(&base, 1, MPI_INT, destination, msg, MPI_COMM_WORLD);
        MPI_Send(&num_rows, 1, MPI_INT, destination, msg,
MPI_COMM_WORLD);
        MPI_Send(&south, 1, MPI_INT, destination, msg, MPI_COMM_WORLD);
        MPI_Send(&north, 1, MPI_INT, destination, msg, MPI_COMM_WORLD);
        MPI_Send(&u[0][base][0], num_rows*Nx, MPI_FLOAT, destination,
msg, MPI_COMM_WORLD);

        printf("Sent to= %d; \t j_index= %d; \t num_rows= %d; \t
south_neighbour= %d; \t north_neighbour=%d\n",
             destination,base,num_rows,south,north);

        base += num_rows;

      }

      /* Collecting and collocating the results */
      for(i=1;i<=nprocs-1;i++)
      {
        source = i;
        msg = DONE;
        MPI_Recv(&base, 1, MPI_INT, source, msg, MPI_COMM_WORLD,
&status);
        MPI_Recv(&num_rows, 1, MPI_INT, source, msg, MPI_COMM_WORLD,
&status);
        MPI_Recv(&u[0][base][0], num_rows*Nx, MPI_FLOAT, source, msg,
MPI_COMM_WORLD, &status);
      }

      /* WRITE FINAL SOULTION*/
      //save(Nx, Ny, &u[0][0][0], "output.dat");


    }
/**************** End of master code *********************************/



/* Slaves code */
  if (myid != MASTER)
    {

#pragma omp parallel private(i,j,k,count)
      {

#pragma omp for private(i,j,k)
      for(k=0;k<2;k++)
        for (i=0; i<Nx; i++)
          for (j=0; j<Ny; j++)
            u[k][j][i] = 0.0;


      /* Receive data from MASTER*/
#pragma omp master
      {
        source = MASTER;
        msg = BEGIN;
```

```
        MPI_Recv(&slave, 1, MPI_INT, source, msg, MPI_COMM_WORLD,
&status);
        MPI_Recv(&base, 1, MPI_INT, source, msg, MPI_COMM_WORLD,
&status);
        MPI_Recv(&num_rows, 1, MPI_INT, source, msg, MPI_COMM_WORLD,
&status);
        MPI_Recv(&south, 1, MPI_INT, source, msg, MPI_COMM_WORLD,
&status);
        MPI_Recv(&north, 1, MPI_INT, source, msg, MPI_COMM_WORLD,
&status);
        MPI_Recv(&u[0][base][0], num_rows*Nx, MPI_FLOAT, source, msg,
MPI_COMM_WORLD, &status);
      }
#pragma omp barrier

#pragma omp sections
      {

#pragma omp section
        {
          for (i=0; i<Nx; i++)
            u[1][0][i]=u[0][0][i];
        }

#pragma omp section
        {
          for (i=0; i<Nx; i++)
            u[1][Ny-1][i]=u[0][Ny-1][i];
        }


#pragma omp section
        {
          for (j=0; j<Ny; j++)
            u[1][j][0]=u[0][j][0];
        }

#pragma omp section
        {
          for (j=0; j<Ny; j++)
            u[1][j][Nx-1]=u[0][j][Nx-1];
        }
      }

      if (base==0)
        start=1; // do not include bottom row or row 0 which is the
boundary
      else
        start=base;

      if (base+num_rows==Ny)
        end= base + num_rows-2;  //do not include top row which is also
the boundary
      else
        end = base + num_rows-1;
```

```
      k=0;

      for(count=0; count<=ITER; count++)
        {

#pragma omp master
          {
            if (south != 0)
            {
              MPI_Send(&u[k][base][0], Nx, MPI_FLOAT, south, NTH,
MPI_COMM_WORLD);
              MPI_Recv(&u[k][base-1][0], Nx, MPI_FLOAT, south, STH,
MPI_COMM_WORLD, &status);
            }
            if (north != 0)
            {
              MPI_Send(&u[k][base+num_rows-1][0], Nx, MPI_FLOAT, north,
STH, MPI_COMM_WORLD);
              MPI_Recv(&u[k][base+num_rows][0], Nx, MPI_FLOAT, north,
NTH, MPI_COMM_WORLD, &status);
            }
          }

#pragma omp barrier


#pragma omp for private(i,j)
          for (j = start; j <= end; j++)
            for (i = 1; i <= Nx-2; i++)
            u[1-k][j][i] = (u[k][j][i+1]+u[k][j][i-
1]+u[k][j+1][i]+u[k][j-1][i])*0.25;

          k = 1 - k;
        }

      }
      eps=errorcheck(start, end, Nx, Ny, &u[k][0][0]);


      /*Send data back to master*/
      destination=MASTER;
      msg=DONE;
      MPI_Send(&base, 1, MPI_INT, destination, msg, MPI_COMM_WORLD);
      MPI_Send(&num_rows, 1, MPI_INT, destination, msg,
MPI_COMM_WORLD);
      MPI_Send(&u[k][base][0], num_rows*Nx, MPI_FLOAT, destination,
msg, MPI_COMM_WORLD);



      printf("Processor number: %d; eps = %6.12f \n", myid, eps);

    }

  end_time=MPI_Wtime();
```

```
  printf ( "Processor number: %d; Total Elapsed time for Hybrid
implementation is  %f  seconds\n", myid, end_time-start_time);


  MPI_Finalize();

}


/*Subroutines*/

void init(int nx, int ny, float *u)
{
  int i,j;
  double hx, hy;

  hx=1.0/(nx-1);
  hy=1.0/(ny-1);


  for (j = 1; j <= ny-2; j++)
    for (i = 1; i <= nx-2; i++)
      *(u+j*nx+i) = (float)(0);  //all the interior points are set to 0

  j=0;
  for (i = 0; i <= nx-1; i++)
    *(u+j*nx+i) = sin(3.14*i*hx);  //bottom boundary

  j=ny-1;
  for (i = 0; i <= nx-1; i++)
    *(u+j*nx+i) = exp(-3.14)*sin(3.14*i*hx);  //top  boundary

  i=0;
  for (j = 0; j <= ny-1; j++)
    *(u+j*nx+i) = (float)(0);  //left boundary

  i=nx-1;
  for (j = 0; j <= ny-1; j++)
    *(u+j*nx+i) = (float)(0);  //right boundary

}

void save(int nx, int ny, float *u1, char *output)
{
  int i, j;
  FILE *f_out;


  f_out = fopen(output, "w");

  for (i = nx-1; i >=0; i--)
    for (j = 0; j <= ny-1; j++)
      fprintf(f_out, "%d %d %6.12f\n", i, j, *(u1+j*nx+i));
  fclose(f_out);
}

float errorcheck(int start, int end, int nx, int ny, float *u)
```

```
{
  int i,j;
  float sum = 0.0;
  float exact;

  double hx, hy;

  hx=1.0/(nx-1);
  hy=1.0/(ny-1);

#pragma omp parallel for private(i,j) reduction(+: sum)
  for (j = start; j <= end; j++)
    {
      for (i = 1; i <= nx-2; i++)
      {
        exact = (sin(3.14*i*hx))*exp(-3.14*j*hy); //exact solution at
the point
        sum = sum + (exact - *(u+j*nx+i))*(exact - *(u+j*nx+i));
//relative error
      }
    }

  return sqrt(sum);
}
/*END OF PROGRAM*/
/************************************************************************
****************************************/
```